DSA Homework 9: Dynamic Programming

David Tarazi

April 11, 2020

1 Minimum Edit Distance

Given two strings, how can we find the minimum number of edits to make the two strings equal? To approach this with dynamic programming we must consider every "case." In this problem a case would be like comparing every substring of str1 to every substring of str2. We know that the real question is when the two substrings are actually the complete strings, so we know we will be building up to compare the full length of both strings. To approach this, we can use the following equations where X(n, m) is the function, and n and m are the two lengths of the strings that we are considering. We know we are finished when n = full length of str1 and m = full length of str2. We essentially have two options: if the two characters match, then we know that they don't impact the edit distance and we can move back a character without changing the minimum edit distance. Otherwise, we must either make an insertion, deletion, or replacement and find the minimum of those operations of the substring before.

$$X(0, m) = m \tag{1}$$

$$X(n, 0) = n \tag{2}$$

The starting conditions above are since there will be m or n insertions/deletions to make the strings equal. Below is the recurring equation.

$$X(n, m) = \begin{cases} X(n-1, m-1) & \text{str1[n] == str2[m]} \\ 1 + min(X(n-1, m), X(n, m-1), X(n-1, m-1)) & \text{str1[n] != str2[m]} \end{cases} \tag{3}$$

And we will terminate in the case of $X(n = length(str1), m = length(str2))$ since we then have the full strings. We know this is correct because for the base case we have a base value of how many operations for any subset of the string against 0 to return a minimum distance and this is set. For each recurring case, we either don't care about the letter if they are equal, or we are going to continuously take a minimum going through the cases to find the minimum operations. If there were a case where the number of operations was not the minimum, then the algorithm hadn't accurately calculated or hadn't checked every possibility recursing back, creating a contradiction.

We know the runtime for this algorithm will be O(n*m) since we could possibly have to check all combinations which would be equal to the length of the strings multiplied together.

2 Wildcard Matching

If we assume we are filling every possible path from left to right, we have a couple of cases here. Either we have reached a point where s1 and s2 are both a character, we must compare those values. However, if we reach a point where s2 is "*" then we must take one of two paths. The first path would be to assume that the "*" will take the current value of the s1 and the second would be to assume that it stops taking the value and moves along in s2. Given these two cases, we can move along until there is an error or the length of the strings all run out and there is a match. When we reach a case where we have traversed both strings fully, we can terminate and if that value is still true, we have reached a match, otherwise, there was some sort of break in the path that signals a mismatch. If we call our equation W(n, m) where n is the index of s1 and m is the index of s2, we can use the following equations below. Starting condition:

$$W(0,0) = True \tag{4}$$

Otherwise we have a couple of options.

$$W(n,m) = \begin{cases} W(n-1,m-1) & \text{s1[n] == s2[m]} \\ W(n-1,m)||W(n,m-1) & \text{s2[n] == "*"} \end{cases} \tag{5}$$

Where the case of $W(n-1,m)$ highlights when the "*" is actually taking the place of a missing character in s1, and $W(n,m-1)$ would indicate that the "*" is either just an empty string or is finished taking the values of s1 and can move on. We know this is correct because if the value is True, then a certain substring matches with another substring and by traversing through these True or False matches by our recurring equation's rules will allow us to find whether or not the path has broken.

The runtime for this function would be O(n*m) since there are that many possibilities for matches that we would have to check through in order to find a match or break.